How to design a Linux kernel interface

Michael Kerrisk
man7.org Training and Consulting
http://man7.org/training/

15 October 2014
Düsseldorf, Germany
Is there a kernel developer in the house?
This is going to be one of those talks
Who am I?

- I’m going to sound like your mother
- But actually:
  - Maintainer of Linux *man-pages* project since 2004
    - Documents kernel-user-space and C library APIs
    - 12,000 commits, 157 releases, author/co-author of 330+ of 970+ pages in project
  - Quite a bit of design review of Linux APIs
  - Lots of testing, lots of bug reports
  - Author of a book on the Linux programming interface
  - IOW: looking at Linux APIs a lot and for a long time
I’m talking more about process than technical detail
Outline

1 The problem
2 Think outside your use case
3 Unit tests
4 Specification
5 The problem of the feedback loop
6 Write a real application
7 A technical checklist
8 Doing it right
Outline

1 The problem
2 Think outside your use case
3 Unit tests
4 Specification
5 The problem of the feedback loop
6 Write a real application
7 A technical checklist
8 Doing it right
Implementation of APIs is the lesser problem

(Performance can be improved later; bugs are irritating, but can be fixed)
API design is the big problem
Why is API design a problem?

- Hard to get right
- (Usually) can’t be fixed
  - Fix == ABI change
  - User-space will break
  - and worse...

- You’ll stop Linus smiling
- And...
Thousands of user-space programmers will live with your (bad) design for decades
Many kinds of APIs

- Pseudo-filesystems (/proc, /sys, /dev/mqueue, debugfs, configfs, etc.)
- Netlink
- Auxiliary vector
- Virtual devices
- Signals
- System calls ⇐ focus, for purposes of example
  - (4 new syscalls in 3.17!)
- `ioctl()`, `prctl()`, `fcntl()`, and other multiplexor syscalls
1 The problem
2 Think outside your use case
3 Unit tests
4 Specification
5 The problem of the feedback loop
6 Write a real application
7 A technical checklist
8 Doing it right
Example: POSIX messages

- POSIX MQs: message-based IPC mechanism, with priorities for messages
  - \texttt{mq\_open()}, \texttt{mq\_send()}, \texttt{mq\_receive()}, ...
  - Linux 2.6.6

- Usual use case: reader consumes messages (nearly) immediately
  - (i.e., queue is usually short)

- Kernel developers coded for usual use case
Example: POSIX messages

- Linux 3.5: a vendor developer raises ceiling on number of messages allowed in MQ
  - Raised from 32,768 to 65,536 to serve a customer request
- I.e., customer wants to queue masses of unread messages
- Developer notices problems with algorithm that sorts messages by priority
  - Approximates to bubble sort(!)
  - Will not scale well with (say) 50k messages in queue...
- Among a raft of other MQ changes, developer fixes sort algorithm
When designing APIs, remember:

User-space programmers are endlessly inventive
And they outnumber you
10,000 to 1
Morale 1: try to imagine the ways in which an army of inventive user-space programmers might (ab)use your API
Is this such a big deal?

A performance bug got found and fixed. So what?

(but there’s more…)
The 3.5 MQ changes broke user space in at least two places

- **Introduced hard limit of 1024 on queues_max**, disallowing even superuser to override
  - Fixed by commit f3713fd9c in 3.14, and in stable
- **Semantics of value exported in /dev/mqueue QSIZE field changed**
  - Now includes overhead bytes
  - [http://thread.gmane.org/gmane.linux.man/7050](http://thread.gmane.org/gmane.linux.man/7050)
Morale 2: without unit tests you will screw up someone’s API
Outline

1 The problem
2 Think outside your use case
3 Unit tests
4 Specification
5 The problem of the feedback loop
6 Write a real application
7 A technical checklist
8 Doing it right
Unit tests

- To state the obvious, unit tests:
  - Prevent behavior regressions in face of future refactoring of implementation
  - Provide checks that API works as expected/advertised
Regressions happen too often
Examples of regressions

- Inotify **IN_ONESHOT** flag
  - (inotify == filesystem event notification API added in Linux 2.6.13)
  - By design, **IN_ONESHOT** did not cause an **IN_IGNORED** event when watch is dropped after one event
  - Inotify code was refactored during fanotify implementation (early 2.6.30’s)
  - From 2.6.36, **IN_ONESHOT** *does* cause **IN_IGNORED**

- Linux 2.6.12 silently changed meaning of `fcntl()` **F_SETOWN**
  - Change discovered many releases later; too late to fix
    - Maybe some new applications depend on new behavior!
  - ⇒ now we have **F_SETOWN_EX** to get old semantics
Does it do what it says on the tin?

(Too often, the answer is no)
Does it do what it says on the tin?

- Inotify IN_ONESHOT flag
  - Provide **one** notification event for a monitored object, then disable monitoring
  - Tested in 2.6.16; simply did not work
    - $\Rightarrow$ zero testing before release...

- Inotify event coalescing
  - Successive identical events (same event type on same file) are combined
    - Saves queue space
  - Before Linux 2.6.25, a new event would be coalesced with item at *front* of queue
    - I.e., with oldest event rather than most recent event
    - Clearly: minimal pre-release testing
Does it do what it says on the tin?

- *recvmsgs()* `timeout` argument
  - Syscall to receive multiple datagrams, added in 2.6.33
  - `timeout` added late in implementation, after reviewer suggestion

- **Apparent** concept: place `timeout` on receipt of complete set of datagrams

- **Actual** implementation: `timeout` tested only after receipt of each datagram
  - Renders `timeout` useless...

- Clearly, no actual testing of implementation

- Also, confused implementation with respect to use of EINTR error after interruption by signal handler
  - [Link](http://thread.gmane.org/gmane.linux.kernel/1711197/focus=6435)
Probably, all of these problems could have been avoided if there were unit tests.
Writing a new kernel-user-space API? ⇒ include unit tests

Refactoring code under existing API that has no unit tests? ⇒

please write some
Where to put your tests?

- Historically, only real home was LTP (Linux Test Project), but:
  - Tests were out of kernel tree
  - Often only added after APIs were released
  - Coverage was only partial

- New (2014) `kselftest` project will hopefully improve matters:
  - In-tree tests
  - Paid maintainer: Shuah Khan
  - Wiki: [https://kselftest.wiki.kernel.org/](https://kselftest.wiki.kernel.org/)
  - Mailing list: `linux-api@vger.kernel.org`
But, how do you know what to test if there is no specification?
Outline

1 The problem
2 Think outside your use case
3 Unit tests
4 **Specification**
5 The problem of the feedback loop
6 Write a real application
7 A technical checklist
8 Doing it right
“Programming is not just an act of telling a computer what to do: it is also an act of telling other programmers what you wished the computer to do. Both are important, and the latter deserves care.”

Andrew Morton, March 2012
Fundamental problem behind (e.g.) `recvmmmsg()` timeout bugs:

no one wrote a specification during development or review
A test needs a specification

`recvmsmsg()` timeout argument needed a specification; something like:

- The `timeout` argument implements three cases:
  1. `timeout` is NULL: the call blocks until `vlen` datagrams are received.
  2. `timeout` points to `{0, 0}`: the call (immediately) returns up to `vlen` datagrams if they are available. If no datagrams are available, the call returns immediately, with the error EAGAIN.
  3. `timeout` points to a structure in which at least one of the fields is nonzero. The call blocks until either:
     - (a) the specified timeout expires
     - (b) `vlen` messages are received

  In case (a), if one or more messages has been received, the call returns the number of messages received; otherwise, if no messages were received, the call fails with the error EAGAIN.

- If, while blocking, the call is interrupted by a signal handler, then:
  - if 1 or more datagrams have been received, then those datagrams are returned (and interruption by a signal handler is not (directly) reported by this or any subsequent call to `recvmsmsg()`.
  - if no datagrams have so far been received, then the call fails with the error EINTR.
Specifications help

Specifications have numerous benefits:

- Provides target for implementer
- Without specification, how can we differentiate implementer’s *intention* from actual *implementation*
  - IOW: how do we know what is a bug?
- Allow us to write unit tests
- Allow reviewers to more easily understand and critique API
  - ⇒ will likely increase number of reviewers
Where to put your specification?

- At a minimum: in the commit message
- To gain good karma: a *man-pages* patch
A well written man page really can be a test specification for finding real bugs:

- `utimensat()`:

- `timerfd`:
  http://thread.gmane.org/gmane.linux.kernel/613442
Outline

1 The problem
2 Think outside your use case
3 Unit tests
4 Specification
5 The problem of the feedback loop
6 Write a real application
7 A technical checklist
8 Doing it right
The problem

- Probably 6+ months before your API appears in distributions and starts getting used in real world
- Worst case: only then will bugs be reported and design faults become clear
- But that’s too late...
  - (Probably can’t change ABI...)
- Need as much feedback as possible **before** API is released
Must radically shorten worst case feedback loop

⇒

Publicize API design as widely + early as possible
Shortening the feedback loop

Ideally, do all of the following before API release:

- Write a detailed **specification**
- Write **example programs** that fully demonstrate API
- Email relevant mailing lists and, especially, relevant people
- CC `linux-api@vger.kernel.org`
  - As per Documentation/SubmitChecklist...
  - Alerts interested parties of API changes:
    - C library projects, man-pages, LTP, trinity, kselftest, LSB, tracing projects, and user-space programmers
- For good karma + more publicity: write an LWN.net article
  - Good way of **reaching end users** of your API
    - Ask readers for feedback
  - [http://lwn.net/op/AuthorGuide.lwn](http://lwn.net/op/AuthorGuide.lwn)
Of course

- Of course, you’d only do all of this if you wanted review and cared about long-term health of the API, right?
- My suspicion: in some case implementers actively avoid these steps, to minimize patch resistance
- Subsystem maintainers: watch out for developers who avoid these steps
Outline

1. The problem
2. Think outside your use case
3. Unit tests
4. Specification
5. The problem of the feedback loop
6. Write a real application
7. A technical checklist
8. Doing it right
Example: inotify

- Filesystem event notification API
  - Detect file opens, closes, writes, renames, deletions, etc.
- A Good Thing™...
  - Improves on predecessor (dnotify)
  - Better than polling filesystems using readdir() and stat()
- But it should have been A Better Thing™
Writing a “real” inotify application

- Back story: I thought I understood inotify
- Then I tried to write a “real” application...
  - Mirror state of a directory tree in application data structure
  - 1500 lines of C with (lots of) comments
  - Written up on LWN (https://lwn.net/Articles/605128/)
- And understood all the work that inotify still leaves you to do
- And what inotify could perhaps have done better
The limitations of inotify

Two among several tricky problems when using inotify:

- Event notifications don’t include PID or UID
  - Can’t determine who/what triggered event
  - It might even be you
  - *Why not supply PID / UID, at least for privileged programs?*

- Monitoring of directories is not recursive
  - Must add new watches for each subdirectory
    - (Probably unavoidable limitation of API)
  - Can be expensive for large directory tree ⇒ see next point
File renames generate MOVED_FROM+MOVED_TO event pair

- Useful: provides old and new name
- But:
  - Items are not guaranteed to be consecutive
  - No MOVED_TO if target directory is not monitored
  - Matching MOVED_FROM+MOVED_TO pairs must be done heuristically and is unavoidably racey
  - Matching failures ⇒ treated as tree delete + tree re-create (expensive!)

*User-space handling would have been much simpler, and deterministic, if MOVED_FROM+MOVED_TO had been guaranteed consecutive by kernel*
Only way to discover design problems in a new nontrivial API is by writing a complete, real-world application

(before the API is released in mainline kernel...)

API limitations should be rectified, or at least clearly documented, before API release...
1 The problem
2 Think outside your use case
3 Unit tests
4 Specification
5 The problem of the feedback loop
6 Write a real application
7 A technical checklist
8 Doing it right
A few technical points that frequently come up in Linux API design
New system calls should have a flags argument

- Bit-mask argument that can be used to extend syscall later
- Default question: is there a reason not to have a flags argument?
- A few examples of many past failures, and their fixes:
  - `futimesat() ⇒ utimensat()`
  - `epoll_create() ⇒ epoll_create1()`
  - `inotify_init() ⇒ inotify_init1()`
  - `renameat() ⇒ renameat2()`
  - And many more
- https://lwn.net/Articles/585415/
Undefined arguments and flags must be zero

- APIs should ensure that reserved/unused arguments and undefined bit flags are zero
  - EINVAL error
  - Allows user-space to test if feature is supported
- Failing to do this, allows applications to pass random values to args/masks
  - Many historical syscalls failed to do this check
- Those applications may fail when future kernels define meanings for those arguments/bits
- Conversely: you may not be able to define meanings, because user-space gets broken
  - (This has happened)
  - https://lwn.net/Articles/588444/
File descriptors syscall should support O_CLOEXEC

- Causes file descriptor (privileged resource) to be closed during `exec()` of new program

- Historical pattern

  ```
  fd = open(pathname, ...);
  flags = fcntl(fd, F_GETFD);
  flags |= O_CLOEXEC;
  fcntl(fd, F_SETFD, flags);
  ```

- Multithreaded programs have a race...
  - If another thread does `fork() + exec()` in middle of above steps, FD leaks to new program

- 2.6.27, + 2.6.28 added raft of replacements for existing syscalls to allow O_CLEXEC to be set at FD creation time
  - E.g., `epoll_create1()`, `inotify_init1()`, `dup3()`, `pipe2()`

- New system calls that create FDs should support O_CLOEXEC
Timeouts on blocking system calls should be absolute

- Relative timers are subject to creep on restart after interruption by signal handler
  - (Because each restart can oversleep)
- Support absolute timeouts on `CLOCK_MONOTONIC` clock
Avoid extending multiplexor system calls

- Disfavor adding new commands to existing multiplexor syscalls
  - `prctl()`, `fcntl()`, `ioctl()`
- No type checking of arguments
- Becomes messy when you later decide to extend feature with new options
Capabilities

- General concept:
  - Divide power of root into small pieces
  - Replace set-UID-root programs with programs that have capabilities attached
  - Less harm can be inflicted if program is compromised
- The problem for kernel developers: what capability should I use for my new privileged operation?
  - Read capabilities(7)
  - Choose a capability that governs similar operations
  - Or, if necessary, devise a new capability
  - Don’t choose CAP_SYS_ADMIN
    - “The new root”
    - 1/3 of all capability checks in kernel are CAP_SYS_ADMIN
    - https://lwn.net/Articles/486306/
- Send in a man-pages patch for capabilities(7)
64-bit arguments and structure fields

- Take care when dealing with 64-bit arguments and structure fields
  - Daniel Vetter, “Botching up ioctls”,
    http://blog.ffwll.ch/2013/11/botching-up-ioctls.html
  - Jake Edge, “System calls and 64-bit architectures”
    http://lwn.net/Articles/311630/
“show me a newly released kernel interface, and I’ll show you a bug”

Yes, bugs are fixable, but...

Bad bugs require user-space to special-case based on kernel version
Outline

1 The problem
2 Think outside your use case
3 Unit tests
4 Specification
5 The problem of the feedback loop
6 Write a real application
7 A technical checklist
8 Doing it right
Doing it right

Jeff Layton, OFD locks, Linux 3.15 (commit 5d50ffd7c31):

- “Open file description locks” (originally: “file-private locks”)
- Fix serious problems with POSIX record locks
- Did everything nearly perfectly, in terms of developing feature
Doing it right

Jeff Layton, OFD locks, Linux 3.15 (commit 5d50fffd7c31):

- Clearly explained **rationale** and changes in commit message
- Provided example programs
- Publicized the API
  - Mailing lists
  - LWN.net article (http://lwn.net/Articles/586904/)
- Wrote a man pages patch
  - (Feedback led to renaming of constants and feature)
- Engaged with glibc developers (patches for glibc headers + manual)
  - Refined patches in face of review
  - Maintainers were unresponsive ⇒ resubmitted *many* times
- Made it all look simple
Thanks!

Slides at http://man7.org/conf/


mtk@man7.org, http://man7.org/training/

*THE LINUX PROGRAMMING INTERFACE*

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK
